

Design Reuse through High-Level Library Mapping[†]

Pradip K. Jha and Nikil D. Dutt

Department of Information and Computer Science
University of California, Irvine, CA 92717-3425, USA

Abstract

We present High-level Library Mapping (HLLM), a technique that permits reuse of complex databook components (specifically ALUs) in architectural synthesis. We describe a dynamic programming formulation of HLLM, demonstrate the versatility of our approach on a variety of libraries and compare HLLM for ALUs with the traditional logic-synthesis approach. Our experiments show that HLLM for ALUs outperforms logic-synthesis in area, delay and runtime, indicating that HLLM is a promising approach for reuse of datapath components in architectural synthesis.

1 Introduction

The motivation for this work stems from the need for incorporating *design reuse* into architectural and high-level synthesis. Since the output of high-level synthesis typically consists of a datapath netlist of generic RTL components and a state sequencing table, there is the important back-end task of realizing the generic datapath components using existing libraries. While module generators and logic synthesis tools can be used to map RTL components into standard cells or layout geometries, they cannot effectively reuse the data book libraries of higher-level components commonly used in design practice. In this paper, we present *High-Level Library Mapping (HLLM)*, a technique for implementing generic RT-level datapath components with technology-specific RTL library components with the goal of facilitating design reuse.

Design reuse plays an important role in the complete design process: system design typically goes through various phases, starting with prototypes implemented in FPGAs, down to a final design implementation composed of custom or standard library parts. Furthermore, existing outputs of high-level design need to be migrated to new libraries and/or tech-

nologies. In such design scenarios, techniques for high-level library mapping are required to effectively support design reuse.

Design reuse is actively practiced at the logic level in current design methodologies: logic-level designs can be retargeted to technology specific gates and flip-flops using a variety of traditional technology mapping techniques. However, as architectural and high-level synthesis tools mature and move the design process to higher levels of abstraction, there is a need to upgrade design reuse techniques to these higher levels of abstraction. Register-transfer(RT) level components such as ALUs and register-files are good candidates for design reuse, since they are often hand optimized or created using module generators. The acceptance of *high-level design* (i.e., refining the behavior of a design into a netlist of RT-level components) has led to a need for design reuse techniques that can port RT-level designs across various vendor libraries.

Specifically, in this paper we present a high-level library mapping technique for arithmetic-logic units (ALUs) derived from the output of high-level design. An ALU can perform a well-defined set of arithmetic, logic, and comparison functions. ALUs are typically optimized and placed in the library, thus becoming ideal candidates for design reuse. Our approach can reuse ALUs from standard library or from datapath generators or even handcrafted components.

In this paper, we review related work (Section 2), define the high-level library mapping problem for ALUs and describe a dynamic programming formulation for this mapping problem (Section 3), and provide experimental results to validate its effectiveness (Section 4). Specifically, we demonstrate the versatility of this approach by applying HLLM to ALUs drawn from different libraries. We also compare the HLLM approach versus a traditional logic synthesis approach and demonstrate the advantage of HLLM for complex datapath components. We conclude with a summary in Section 5.

[†]This research was supported by SRC contract #93-DJ-146.

2 Related work

Design reuse techniques for architectural and high-level synthesis can be broadly divided into two categories. One school of thought attempts to directly incorporate components from a technology library into the high-level design phase. This approach combines two steps of the high-level design process (design synthesis and technology mapping), and provides a monolithic approach to the reuse of library components. Although this approach can yield good results, it requires a lot of effort in tuning the synthesis and refinement tools to accommodate variances in RT-level technology components. Research efforts described in [1] [8] [11] [13] fall into this category.

The other school of thought believes in a two-tier process. First, the design refinement phase maps the behavior into an intermediate (generic component) level. Second, this intermediate (generic) RT design is mapped into a final implementation of technology library parts through logic-level technology mapping techniques. Each RT component's functionality can be described using Boolean equations for the transformation of the inputs into outputs. These equations can then be mapped to low-level technology-specific components such as gates, flip-flops and latches. Hence, technology mapping is done at the logic level on large flat designs, which requires DAG matching by detecting isomorphism of large subgraphs [3] [10] [12] [14]. This complicates the task of interfacing to a given RT library that may consist of large components at the MSI and LSI levels.

[4] investigated the relationship between logic-level and high-level synthesis and illustrated some basic tradeoffs. At slightly higher level, [7] maps a RT component by functionally/structurally decomposing it to MSI-level building blocks from a technology library. [15], on the other hand, combines the functional decomposition and logic-level mapping techniques along with microarchitectural optimization to realize a netlist of RT level components. At the system-level, MICON [2] tries to reuse off-the-shelf system-level parts such as processors, memories and peripherals to build a single-board computer system. Our **high-level library mapping** approach realizes a RT-level component with another RT-level component of similar complexity and therefore complements these previous approaches.

3 Problem definition and approach

The high-level library mapping problem for ALUs is based on functional specifications of the source and

target components, which are compared with respect to a “canonical” functional representation to derive an effective mapping result. When the functionality of the target component does not exactly match that of the source component, the target component may need to be padded with extra (glue) logic. Figure 1 illustrates this high-level mapping approach between a source and a target ALU.

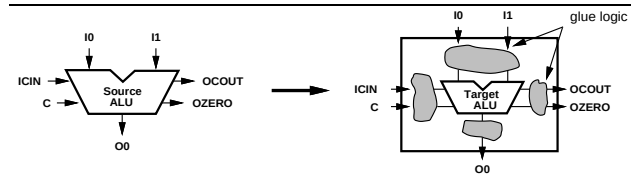


Figure 1: High-level library mapping of an ALU

We therefore define the high-level library mapping problem in terms of a *Source component* (S), a *Target component* (T) and a set of *Mapping rules* (R) that maps the source component onto the target component. In order to establish equivalence between the source and the target components, each component is described in terms of a set of RT-functions that are defined using a canonical representation of the component. A mapping rule in R describes an alternative for implementing a function in S using a function in T ; each source function can potentially be implemented by different target functions. The task of high-level library mapping, then, reduces to selecting a set of rules, one for each function in source component S , that realizes the best mapping of S on T with respect to the cost function (e.g., area or delay).

Figure 2 illustrates our overall approach to high-level library mapping. The source component (S), the target component (T) and the mapping rule database (R) comprise inputs to the system. The mapping algorithm implements S on the target component T by selecting a set of mapping rules that yield the minimum overall cost. The output of the system is an implementation of the source component S on the target component T , along with some glue logic. In this paper we restrict our domain of components to the HLLM of ALUs. We illustrate each of the steps in the approach by walking through an example.

3.1 Representation of library components

The source and the target components are represented in terms of a canonical ALU and the set of functions they perform. A canonical ALU can perform any subset of 27 functions: 5 arithmetic (ADD, SUB, RSUB, INC, DEC), 16 logic (all Boolean functions of 2 variables) and 6 comparison

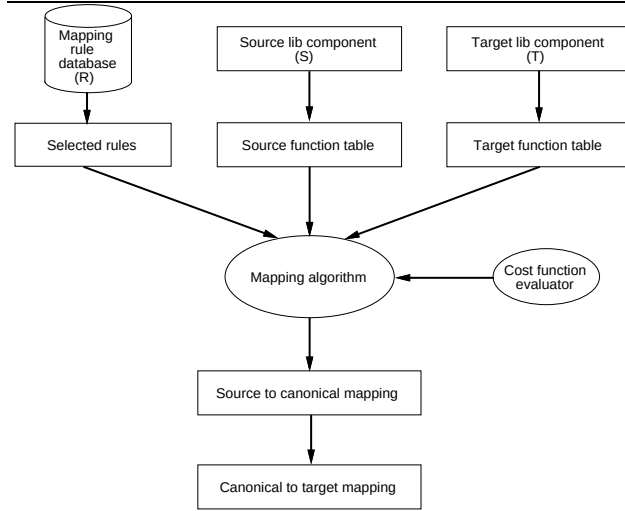


Figure 2: Overall approach to high-level library mapping

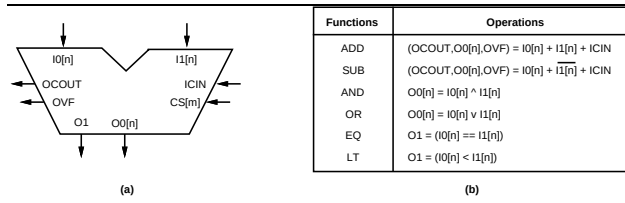


Figure 3: The canonical ALU: (a) Port descriptions (b) Sample functions

(EQ, NEQ, GT, GEQ, LT, LEQ) functions. The canonical ALU as shown in Figure 3(a) has two primary inputs ($I0$ and $I1$), a primary output ($O0$), carry input ($ICIN$), carry output ($OCOUT$), comparison output ($O1$), overflow signal (OVF) and control signals (CS). Note that a source or a target ALU may not have all these ports; the mapping algorithm must accommodate such cases.

The canonical ALU is characterized by its functions that define the relationship between its inputs and outputs. Note that a function need not use all the ports of the ALU. Figure 3(b) shows the canonical representation for some sample ALU functions. The arithmetic functions (e.g., *ADD*, *SUB*) make use of the following ports: $I0[n]$, $I1[n]$, $O0[n]$, $ICIN$, $OCOUT$ and OVF . Logic functions such as *AND* and *OR* use only the primary inputs ($I0[n]$, $I1[n]$) and primary output ($O0[n]$). The comparison functions (e.g., *EQ*, *LT*) on the other hand use primary inputs ($I0[n]$ and $I1[n]$) and $O1$.

An ALU library component (e.g., *S* or *T* in our problem) is described by its port list and its set of functions. Each function for a source or target library component is a variant of one of the 27 canonical ALU functions, and is described by its corresponding canon-

ical function and the Boolean relationship between the ports of the library component and the ports of the canonical component. The Boolean relation between the library component ports and the universal component ports represents the glue logic and is described in a tabular fashion. Figures 4 and 5 illustrate the representation of the source and target components respectively for our walk-through example. The source component (*S*) in Figure 4 is an arithmetic unit that can perform three functions: *ADD*, *SUB*, *RSUB*. The target component (*T*) in Figure 4 can perform two functions: *ADD*, *SUB*.

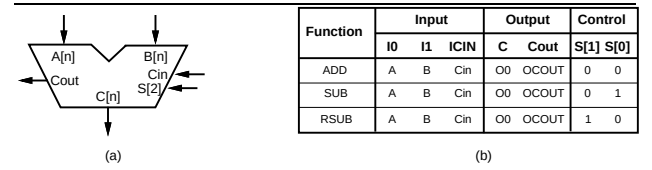


Figure 4: Example source component: (a) Port description (b) Function table

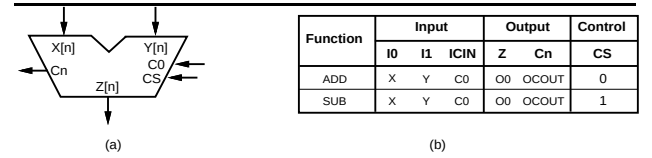


Figure 5: Example target component: (a) Port description (b) Function table

3.2 Mapping rule representation

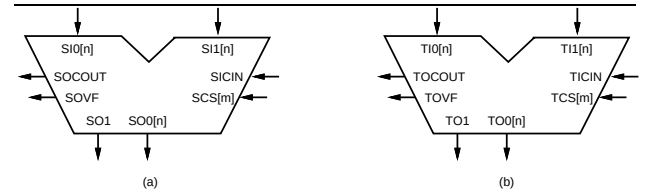


Figure 6: Port names: (a) Source canonical ALU (b) Target canonical ALU

A mapping rule describes a method for implementing one canonical function using another canonical function. Let *SF* and *TF* be the source and the target canonical functions respectively, and let the source and target canonical components (*CS* and *CT*) have the port names shown in Figure 6. A mapping rule then describes the mapping of *CS* ports onto *CT* ports such that *SF* is implemented using *TF*.

Figure 7 lists some sample rules applicable to our walk-through example. Each rule describes an implementation of a source function using a target func-

Rule name	Source function	Target function	Input ports			Output ports		
			Ti0	Ti1	TiCIN	SOcOUT	SOVF	SO1
AA1	ADD	ADD	Si0	Si1	SiCIN	TOcOUT	TOVF	-
AS1	ADD	SUB	Si0	$\overline{Si1}$	SiCIN	TOcOUT	TOVF	-
SA1	SUB	ADD	Si0	$\overline{Si1}$	SiCIN	TOcOUT	TOVF	-
SS	SUB	SUB	Si0	Si1	SiCIN	TOcOUT	TOVF	-
RA1	RSUB	ADD	$\overline{Si0}$	Si1	SiCIN	TOcOUT	TOVF	-
RS	RSUB	SUB	Si1	$\overline{Si0}$	SiCIN	TOcOUT	TOVF	-

Figure 7: Selected set of rules for mapping example

tion and indicates the port mappings required to implement the mapping. The table shows two mapping rules for each source function. For example, the source *ADD* function could be implemented using the target *ADD* function (rule “AA1”), or with the target *SUB* function (rule “AS1”). The input and output port entries indicate the connectivity and additional logic required to implement the mapping rule. For instance, the last rule “RS” implements the source function *RSUB* (reverse *SUB*) with the target function *SUB* by switching the left and right inputs.

Not only are there multiple mapping rules for each source function, but the selection of mapping rules for various source functions are also interdependent. For example, if we decide to use the rule “AS1” (Table 7) for mapping source function *ADD*, rule “SA1” for source function *SUB* would lead to an efficient implementation since it shares the factor $\overline{Si1}$ for the right primary input. Thus a strategy is required to select a mapping rule out of multiple alternatives for each source function. Section 3.5 presents a mapping algorithm that performs this selection of mapping rules in an efficient manner.

3.3 The cost function

A good cost function captures the important characteristics of an efficient source-to-target component mapping. We use a cost function based on two criteria: (a) an area metric, represented by the *gate-count* of the hardware overhead, and (b) a delay metric, represented by the worst case delay or *max-delay* of the generated design.

3.4 Source to target mapping

Referring back to Figure 2, the mapping algorithm implements *S* onto a canonical ALU (*C*) using only those functions that are present in *T*. This mapping is achieved by finding a set of rules, one for each source function, such that the cost of extra hardware (i.e., gate count or delay) is minimized. The set of selected rules provides the connectivity between the ports of *S* and *C*. Figure 8(a) shows one such solution for our

walk-through example in terms of selected set of rules and the canonical implementation.

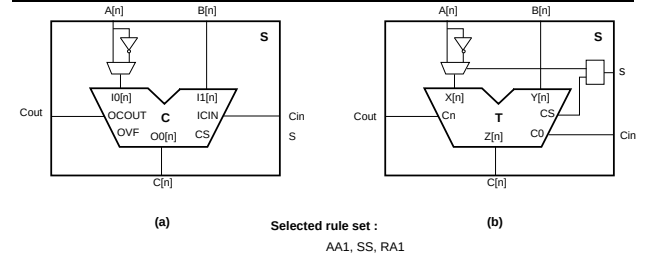


Figure 8: Mapped design: (a) Mapping onto canonical component(*C*) (b) Mapping onto target component(*T*)

The final step maps the canonical ALU (*C*) onto the target component (*T*). This is a simple process since we restrict *T* to be quite similar to *C*. This step connects the ports of *C* to the ports of *T*. Figure 8(b) shows the final implementation.

3.5 Dynamic programming algorithm

We formulated high-level library mapping as a dynamic programming [5] problem that builds up partial solutions (mapping of a subset of source functions) in a bottom-up fashion. It starts with the mapping for the subsets of single functions, followed by mapping for subsets of two functions and continues in this fashion until it has a mapping for the entire set of source functions. Each partial mapping for a subset of the functions is stored in a table and is subsequently used for building the partial solutions for subsets of bigger size.

Algorithm 3.1 : Dynamic programming algorithm

INPUT: $f(S)$, $f(T)$, $r(ST)$, k .

OUTPUT: A set of rules, one for each function in $f(S)$.

1. Order the functions $\in f(S)$.

2. for $i = 0$ to $nS-1$ do

2.1 Table[i, i] = k -best way of mapping f_i .

3. end for

4. for $i = 1$ to $nS-1$ do

4.1 for $j = 1$ to $(nS-i)$ do

4.1.1 CreateEntry($j-1, i+j-1$).

4.2 end for

5. end for

Algorithm 3.1 shows our dynamic programming formulation. Since the number of partial solutions increases exponentially with the size of the function set, we restrict ourselves to k partial solutions for each subset (also called the bucket size). In this algorithm, $f(S)$, $f(T)$ and $r(ST)$ represent the set of functions in the source component, the set of functions in the tar-

get component and the set of mapping rules respectively. The algorithm builds up a table of partial solutions in a bottom-up fashion. This table is indexed by number of source functions (nS) both for row and column. An entry $table(i, j)$ represents partial solutions for source function i to source function j . This algorithm iteratively fills up the table with partial solutions. It starts by filling the diagonal entries by generating the mapping for single functions. Then it fills up the entries corresponding to two function sets and so on. The final solution is given by the top-row and right-most column. The function *CreateEntry* creates the list of partial solutions for a set of source functions using the partial solutions generated so far.

Algorithm 3.2 : *CreateEntry*(row,col)

1. $min_k = 0$.
2. **for** $i = 0$ to $(col-row-1)$ **do**
 - 2.1 $min_k = k$ -best of $combine(table(row, row+i), table(row+i+1, col))$.
3. **end for**

The complexity of the algorithm is $O(n^3k^2)$, where n is the number of source functions and k is the bucket size. Further details on the algorithm and our approach can be found in [9].

4 Experimental results

We performed two sets of experiments to validate our approach. The first set of experiments examines the comprehensiveness of high-level library mapping across different databook and generator libraries. The second set of experiments compares our approach against the traditional approach using logic synthesis. Detailed results are presented in [9]; we present a summary of the results here.

4.1 Comprehensiveness

In order to evaluate the comprehensiveness of our HLLM approach, we chose a variety of source and target libraries for the ALUs. For the lack of space, we present a very brief summary of the experiment and the conclusion. Detailed results are available in [9].

In this experiment, we considered ALUs from four different libraries: CASCADE, GENUS[7], VTI(VDP300) and AMD(Am2901). Starting from a simple uni-function adder, the most complex ALU had 32 functions that covered functions from different categories: arithmetic, logic and comparison. The ALUs were also of various bit-widths. We observed that our approach is quite versatile in the sense that it can map ALUs from wide a variety of libraries and ALUs

with diverse complexity in terms of bit-width, number of functions and set of functions. The dynamic programming algorithm generates designs of high quality, often optimal designs, in the order of minutes.

4.2 Goodness

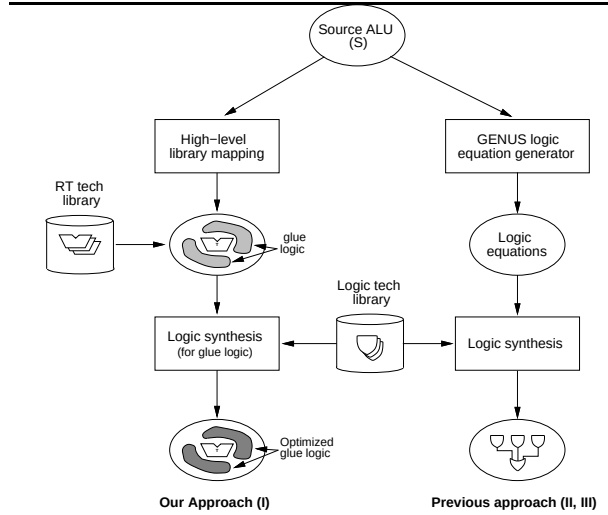


Figure 9: Experimental set-up for comparative study

To illustrate the advantage of HLLM, we performed experiments that compare designs produced by our approach against those produced by a commercial logic synthesis tool[14]. For both cases, the target components were ALU macros from the Synopsys DesignWare library[6]. For each source component, we first map this component onto a macro in [6] library and glue logic around it using our approach. Then we pass the resultant design through logic synthesis to optimize the glue logic. We present metrics for the designs (I) arrived by this path (our approach). In the next phase of the experiment, we generate the logic equations for the input source component and directly map these onto gate-level cells using logic synthesis. Logic synthesis is tried with two levels of optimizations: low (II) and medium (III). For each of these designs, we present design metrics: gate-count, max-delay and run-time and compare them against the metrics for designs generated by our approach (I).

The table in Figure 10 shows metrics for designs that are optimized for area (in terms of gate-count). Similar results were obtained for delay-optimized designs. Columns 2 to 4 in this table describe the source and target components. The “Metrics” column indicates, for each mapping, the area, delay and run-time for various approaches. The column titled “Our Approach” reports area, delay and run-time for the

Ex	Source component (GENUS)	Target component (Designware)	Bw/ Style	Metrics	Our approach I	Logic synthesis (Synopsys)			
						II	%diff	III	%diff
1	ADD,SUB,RSUB	ADD,SUB	32/ cla	Area(2-inp)	962	2450	154	2296	139
				Delay(ns)	11.68	31.92	173	13.60	16
				Run-time(min)	13.90	19.00	37	38.20	175
2	ADD,SUB,RSUB,INC,DEC	ADD	27/ ripple	Area(2-inp)	421	2473	487	1502	257
				Delay(ns)	36.53	64.85	78	32.00	-12
				Run-time(min)	6.03	22.33	270	47.98	696
3	ADD,SUB,RSUB,OR,AND,LINH,LXOR,XNOR	ADD,SUB	32/ cla	Area(2-inp)	1237	3285	165	3313	168
				Delay(ns)	14.27	30.57	114	14.39	1
				Run-time(min)	17.40	27.00	55	55.48	219
4	ADD,SUB,AND,NAND,OR,NOR,XOR,XNOR,LID,RID,LNOT,RNOT,ONE,NEQ,GT	ADD,SUB	16/ cla	Area(2-inp)	543	2506	361	2244	313
				Delay(ns)	12.88	17.32	34	14.90	16
				Run-time(min)	11.80	22.31	89	40.90	247

I : low optimization II : low optimization III : medium optimization
Area : in 2-input gates Delay : in nanoseconds Run-time : in minutes on sparc2

Figure 10: Our approach verses logic synthesis

designs generated by our approach (I). The last two columns in Figure 10 present metrics for designs produced by the logic synthesis approach with low(II) and medium(III) optimization effort. We also report the percentage difference between design metrics for II and III with respect to our approach (I).

From Figure 10, we observe that except for one case (delay for example 2), the designs from our approach outperform designs produced by the logic synthesis approach. Logic synthesis designs with low optimization (II) are as much as 487% larger, 173% slower and require 270% extra run-time. Even with medium optimization(III), designs are 313% larger, 16% slower at the cost of even larger difference in run-time(696%). Logic synthesis with the high optimization script ran for weeks on these examples and many had to be terminated prior to completion. These results demonstrate the efficiency of HLLM for mapping regularly-structured datapath components.

5 Summary

We presented a novel library mapping approach for the output of architectural synthesis at the RT-level based on functional specifications of the source and target components. High-Level Library Mapping (HLLM) allows us to explore the effects of different libraries on the output of architectural synthesis by reusing components from standard databook libraries, datapath generators or even handcrafted datapath components. We presented a dynamic programming formulation for the HLLM of ALUs, and described its use for generating either area-optimized or delay-optimized designs.

In our experiments, we demonstrated the versatility

of our approach by applying HLLM on ALUs drawn from a wide variety of libraries. We also demonstrated the superiority of our approach over logic synthesis for complex ALU components in all the three metrics: area, delay and runtime. We believe that the HLLM approach needs to complement logic synthesis and traditional technology mapping techniques to bridge the output of architectural synthesis with RT-level databook libraries, module generators and handcrafted components. Future work will investigate such interactions.

6 References

- [1] R. Ang and N. Dutt, "An Algorithm for Allocation of Functional Units from Realistic RT Component Libraries," *Symposium on High-Level Synthesis*, May 1994.
- [2] W. Birmingham, A. Gupta and D. Siewiorek, "The MICON System for Computer Design," *DAC*, 1989.
- [3] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "MIS : A multiple-level logic optimization system," *IEEE Transaction on CAD*, November 1987.
- [4] R. Camposano and L. Trevillyan, "The Integration of Logic Synthesis and High-Level Synthesis," *ISCAS*, 1989.
- [5] T. H. Cormen, C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms," *The MIT Press, Cambridge, Massachusetts*, 1990.
- [6] "DesignWare Databook, Version 3.0," *Synopsys[®] Inc., Mountain View*, December 1992.
- [7] N. Dutt and J. Kipps, "Bridging High-Level Synthesis to RTL Technology Libraries," *DAC*, June 1991.
- [8] W. Guerts, F. Catthoor, and H. De Man, "Heuristic Techniques for the Synthesis of Complex Functional Units," *The European Conference on Design Automation*, 1993.
- [9] P. Jha and N. Dutt, "High Level Library Mapping for Arithmetic Components," *Technical Report 94-15, University of California at Irvine*, 1994.
- [10] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," *DAC*, 1987.
- [11] B. Landwehr, P. Marwedel and R. Domer, "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming," *European Design Automation Conference*, 1994.
- [12] F. Mailhot and G. Micheli, "Technology Mapping Using Boolean Matching and Don't Care Sets," *European Design Automation Conference*, 1990.
- [13] E. Rundensteiner, D. Gajski and L. Bic, "Component Synthesis From Functional Descriptions," *IEEE Transaction on CAD*, September 1993.
- [14] "Design Analyzer Reference Manual, Version 3.0," *Synopsys[®] Inc., Mountain View*, December 1992.
- [15] N. Vander Zanden and D. Gajski, "MILO: A Microarchitecture and Logic Optimizer," *DAC*, June 1988.